ShelbySim: A Transparent, Pedagogy-Oriented Simulator for Computer-Based Systems

Dan Tappan

College of Engineering, Idaho State University
tappdan@isu.edu

Abstract

ShelbySim is a student-friendly, pedagogy-oriented, open-source software system for designing, simulating, and evaluating a wide range of multidisciplinary, computer-based engineering applications. It consists of three top-level, seamlessly integrated and richly expressive components that focus on software, hardware, and a holistic combination of the two in realistic operational contexts. The software component consists of a Java-like object-oriented programming language, Shelby, a full-fledged, transparent compiler with extensive tracing, logging, and inspection capabilities, and a runtime system for executing its programs. The hardware component is for designing and implementing conceptualized representations of embedded systems and computer architectures that support the software. The simulation component provides a flexible, interactive framework for running controlled experiments on the software and hardware. It provides students with raw data for quantitative performance analysis, evaluation, and reporting of their designs. ShelbySim also functions as an appealing tool for demonstrations and recruitment.

1 Introduction

Most engineering students, due to tight curriculum constraints and their own choice, tend to fixate on the material of their major discipline at the expense of a broader, holistic view of how it integrates with other, closely related disciplines [1]. Such overemphasis on depth over breadth can be a disadvantage in the contemporary, multidisciplinary work environment. ShelbySim is a pedagogy-oriented software system that exposes students interactively to the design, simulation, and analysis of a broad range of manageable, real-world engineering applications. It provides a unified, seamlessly integrated framework to investigate how software and hardware interoperate in computer-based, or embedded, systems. It also provides student-friendly data-collection and simulation capabilities that mitigate much of the mundane, distracting overhead in course assignments, thereby giving students more hands-on time for the content and its meaningful analysis.

With the 2006 American Competitiveness Initiative to double the number of science, technology, engineering, and mathematics (STEM) graduates in the United States by 2015, it is now more important than ever to recruit and retain qualified engineering students [2]. To this end, ShelbySim also functions as an attractive recruitment tool.

2 System Organization

ShelbySim consists of three top-level components: software, hardware, and simulation. The software and hardware components primarily address aspects of computer science and electrical/computer engineering, respectively. The simulation component combines the software and hardware components within an operational context to solve a problem.

ShelbySim is open-source software written in Java, with many of its internal components written in its own language, Shelby. Java is platform-independent, which eliminates the course-management complexities of supporting the software on the many computer configurations students have. Java 3D, which is a freely available, platform-independent add-on to Java, manages the three-dimensional graphics, animation, and sound effects for advanced features, especially for the recruitment-targeted demonstration mode.

2.1 Software Component

The software component consists of three subcomponents: the Shelby programming language, its compiler, and a runtime system for interpreting programs written in it.

2.1.1 Language

Shelby is a practical, object-oriented programming language based very closely on Java. Although there are some minor differences, as Table 1 shows, any student familiar with Java should be able to transition easily to it. For the sake of pedagogical clarity in the compiler, it omits some convenient, but not strictly necessary, features like inner classes, generics, and all but the most common packages like container data structures and input/output classes. In support of its hardware role, it adds some lower-level features like built-in binary data types, direct access to hardware flags such as carry-in/out and overflow, and interrupt handling.

abstract	assert	break	case	catch	class
continue	default	delete*	do	else	enum
extends	final	finally	for	friend*	if
implements	import	inline*	instanceof	interface	invariant*
new	package	packaged*	private	protected	public
return	static	subpackaged*	switch	throw	throws
transient	try	void	volatile*	while	

Table 1: Shelby keywords; *not present in Java

Java is a derivative of C++, especially in syntax. For various reasons, its developer, Sun Microsystems, decided to omit certain C++ features. This choice gave rise to different programming practices for the two languages, which are currently the most popular in use. The Shelby language is intended to reflect modern programming practices in general, and, as such, also optionally supports some language features from C++ on pedagogical grounds. The most apparent are pointers (with the dreaded `*` and `&` syntax) and manual memory management (the `delete` keyword), both of which endlessly confound students—and many professionals.

The built-in, primitive data types in Table 2 are based on a 24-bit system. While small by modern standards (16MB of addressable memory), it provides a practical balance between size and complexity, especially when using ShelbySim to study computer architecture and organization. For further simplicity, there are no short and long variants for integer and real data types. Real numbers also use a straightforward, fixed-point representation (i.e., an integer value with a decimal-point position) because modern floating-point methods are arguably beyond the scope of an undergraduate curriculum. This decision incidentally makes for an interesting analytical study between the two representations.

Type	Bits
binary*n*	$1 \le n \le 24$
boolean	1
octet*	3
nibble*	4
byte*	8
char	8
word*	16
int*	24
pointer<*type*>	24
real	24
unit*	24

Table 2: Primitive data types; *unsigned and signed variants

2.1.2 Compiler

A compiler translates a high-level, human-readable language like Shelby into a cryptic, low-level language that runs on actual or emulated hardware. It is a mysterious black box that students—and professionals—take for granted because it hides the countless gory details of this intricate process. Although abstracting away such complexity is desirable in a work setting, it is actually counterproductive in an educational one because these details expose underlying relationships between software and hardware.

The Shelby compiler performs the functions of any comparable compiler, but it does so within the pedagogical framework of ShelbySim. For instance, nearly every aspect of the compilation process, from syntactic and semantic analysis through code generation and optimization, can trace the processing and selectively log details of interest. There are numerous options for enabling or disabling programmatic features, as dictated by the instructor. For example, students can be forced to perform manual memory management by turning off the automatic garbage collector. If they fail to deallocate memory correctly, which is a very common pitfall in C++, the problem and its consequences are made clear.

For efficiency, real-world compilers like `javac` and `g++` perform as many simultaneous compilation tasks as possible. The result is pedagogically negative in two critical respects. First, the compiled target code bears little or no resemblance to the source code the students wrote, which makes it exceedingly difficult for students to see the correspondences at any level. Second, the source code of the compiler itself, which is so vast, complex, and interdependent, makes its nearly impossible to understand and extend it as a course assignment. With the Shelby compiler, it is reasonable to task students to implement new features in the language. It manages this feat by subdividing the compilation process into at least 12 passes, each of which performs generally one clear, concise task:

1. Resolve imports and load source files.
2. Qualify static scopes like packages, classes, interfaces, enumerations, labels, methods, and variable declarations. For example, variable `w` in method `x` of class `y` in package `z` is designated `z.y.x.w`, which uniquely identifies it throughout the entire system.
3. Perform at least 20 early validation checks on the source files. For example, source files must be in the proper folders, constructor and class names must match, and the `default` case must be last in a `switch` statement.
4. Register the definitions of and references to all classes, interfaces, and enumerations. This pass, in combination with 5 and 6, produces a rich graph showing the interrelationships between entities in a program.
5. Register inheritance relationships in classes, interfaces, and enumerations.
6. Register derived methods, constructors, class variables, enumeration members, and interface members resulting from inheritance.
7. Determine the types of all variables in the context of where and how they are used. This pass and 8 clearly enumerate at least 20 special considerations. They also clarify how inheritance and polymorphism really work, which helps students to use these concepts more appropriately.
8. Bind overloaded methods based on the argument types in calls to them.
9. Perform at least 20 late validation checks for valid assignments, casts, contractual obligations, protection violations, and exception handling.
10. Identify and discard unused type definitions, methods, and variables. This pass especially demonstrates applications of the discrete mathematics and graph theory that students must learn, but rarely appreciate.
11. Generate the intermediate code.
12. Optimize the intermediate code. This pass is primarily reserved for extension by advanced students. Real-world optimizations are primarily graduate-level concepts. Employing them severely obfuscates the meaning of a program.

2.1.3 Runtime System

The intermediate code produced by the compiler executes in one of two ways. The most straightforward uses the built-in, software-based runtime system (similar to the Java Virtual Machine™), which interprets the code and performs its corresponding hardware-like actions. The second way uses an actual simulated architecture, which will be covered in the next section.

The runtime system is a relatively simple, virtual computer for the 16MB, 24-bit machine defined in 2.1.1. Running compiled programs this way allows students to focus predominantly on software instead of on hardware. All details that were logged throughout the earlier stages are available, which allows students to see how their programming choices are actually realized. These details are fully crossreferenced, so students can follow the audit trails forward and backward from any point with an interactive browser. Such critical inspection is nearly impossible in a normal computer environment because the details are too complex, convoluted, and hidden.

2.2 Hardware Component

The hardware component plays a role comparable to existing schematic capture and simulation software like MultiSim™ and LogicWorks™. Its distinguishing pedagogical hallmark is reflected in how tightly and seamlessly it integrates with the inspection capabilities of ShelbySim overall to show how software and hardware interact in real time. It is also student-friendly, without overwhelming, mysterious options and advanced, professional features that distract students into focusing more on the tool than on the task it is supposed to help solve.

Interaction is through the customary bimodal approach, where students first design (“capture”) a schematic diagram, then they run a simulation directly on it. This approach supports consistency because the view of the design remains the same in both contexts. The demonstration aspects for recruitment also come into play here.

As a simulator of computer-based systems, ShelbySim emphasizes digital circuits. To this end, all the customary discrete, digital components are available; e.g., logic gates, latches, flip-flops, memory devices, etc. Each component is defined by its own Shelby program, which makes extending components or adding new ones very convenient. For pedagogical flexibility, components are typically abstractions of their real-world counterparts. For example, a 21-input AND gate is likely not available from any manufacturer, but if students need one to play this role in their design, it is a simple matter of defining this configuration. Such abstraction eliminates the hierarchical cascading of multiple, smaller gates, which leads

to messy, disorganized, and confusing designs. In other words, ShelbySim allows students to focus on the design more so than on the low-level details of its realization. This approach is acceptable because ShelbySim, unlike many other tools, is not intended for building the actual circuits in a lab.

Another example of abstraction is the mitigation or elimination of common, troublesome phenomena that complicate designs. For example, in real life, propagation delay, clock skew, fan-out limitation, and so on cannot be avoided, but in a pedagogical environment, they unnecessarily complicate the learning process and frustrate students, who otherwise may have good designs. In the event that such delays are actually necessary or desirable, such as with some sequential logic devices, they can still be modeled.

Although the hardware component is primarily for digital systems, it does incorporate a reasonable (and expandable) selection of common analog and electromechanical components for interfacing and testing purposes. For example:

- Input: switches, buttons, keypads, keyboards, encoders, sensors (distance, temperature, light, rotation, etc.)
- Processing: analog-to-digital and frequency-to-amplitude converters (and their inverses), signal conditioners, serial and parallel communication devices, relays and basic switching transistors, circuit breakers.
- Output: light bulbs, LEDs (single, numeric, alphanumeric), small and full-screen LCD displays, motors, linear actuators, solenoids.
- Testing: oscilloscope, voltmeter, logic analyzer, data logger.

ShelbySim does not yet graphically model the physical realization of devices the way full-featured, professional computer-aided-design packages like SolidWorks™ and LabVIEW™ do. However, some form of their nicer representation will eventually be incorporated for better visual understanding and appeal, especially for demonstrations.

ShelbySim was originally intended for courses in computer architecture and organization. As such, it especially facilitates building and testing any reasonable architecture, from small microcontrollers for embedded systems to 24-bit, pipelined microprocessors with an operating system (Shelbix) for general-purpose computing. The Shelby compiler contains many options for translating its intermediate code to various target architectures (contrived or actual). For example, a microcontroller with a 4-bit data bus is very educational because, with minimal hardware, it effectively demonstrates the operations necessary to fetch and execute machine-code instructions [3]. It does not, of course, directly accommodate a programming language with 24-bit data like Shelby. Two compiler options are available to resolve this situation. The first is the easiest, although not the friendliest or most flexible: it actively prevents the program from exceeding the reduced capabilities of the hardware by halting on any violation.

The second, which requires additional, thoughtful coding in the compiler by students, depending on what they opt to support, transforms problematic operations into multiple steps. For example, an 8-bit transfer can be managed as two sequential 4-bit transfers. In fact, schemes like this abound in hardware, and compilers routinely perform such sneaky transformations without the knowledge of the programmer. ShelbySim exposes this magic to students and forces them to weigh design decisions and limitations. For example, this 8-bit transform might be reasonable because it accommodates the standard byte data type (a definite advantage) with only a doubling of the transfer time (an acceptable disadvantage). On the other hand, extending it to the full 24 bits may not satisfy a cost-benefit analysis.

2.3 Simulation Component

The simulation component provides the operational framework for evaluating how well the software and hardware work together as a solution to a reasonable engineering problem in an academic environment. It consists of simulation and evaluation stages.

2.3.1 Simulation

Simulation is the process of running the software on the hardware, manipulating the inputs, and observing the outputs under various operating conditions. Manipulation occurs in two modes: deterministic and stochastic. Deterministic mode involves changing the inputs either interactively or batchwise through data files. The former is convenient for initial testing and to observe immediate cause-and-effect relationships. The latter is more appropriate for running careful sequences of events multiple times, where students tweak their design between runs to debug or evaluate it. In either case, students must explicitly choose discrete values for all the inputs. In contrast, stochastic mode runs a non-interactive Monte Carlo simulation, where inputs are discrete values, events, or ranges, any of which may have associated probabilities. It is covered in the next section.

The simulated application in Figure 1 is a closed-loop feedback system that models a speed controller, much like a basic automobile cruise control:

- Power supply *P* provides power as a unitless 100% potential. In this conceptualized model, current, voltage, battery drain, etc. are not considered, although they could be.
- Breaker *B* limits the power from *P*. Its maximum threshold is set as a property (0–100%).
- Motor *M* has a power input (0–100%), an inertial load input (0–100%), and a built-in RPM output (0–999). The relationship between power, load, RPM, acceleration, response time, and so on is defined by a Shelby program. This interface is built into the motor itself here to avoid having to model irrelevant complexities like a rotating shaft and its encoder.
- Linear slider *T* manually throttles the expected RPM. It is calibrated in percent, not RPM. It also has an input from the controller that adjusts it automatically to maintain the expected RPM as the load changes.
- Linear slider *L* manually proportions the load to the motor. It is also calibrated in percent.
- Pushbutton *S* latches the current RPM value to maintain.
- Display D_A shows the actual RPM of the motor.
- Display D_E shows the expected RPM of the motor, as latched by *S*.
- Controller *C* runs a Shelby program that maps the inputs to the output. As the simulation runs, students can change any of the manual inputs *T*, *S*, and *L* and watch the effect on *M* via D_A, D_E., and *T* (which is also an output). Every data element is automatically logged in real time for analysis.

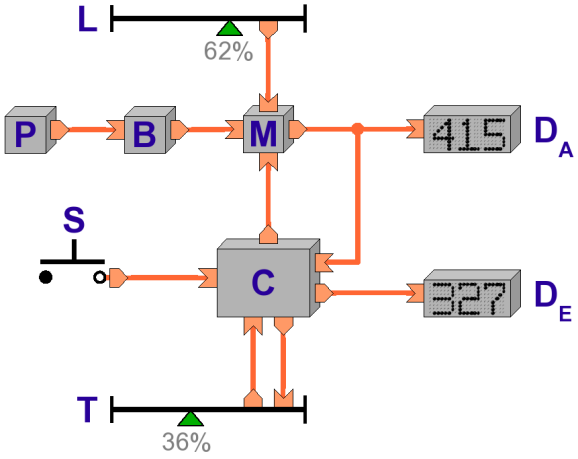

Figure 1: Example schematic for cruise control[1]

As a conceptualized system, explicitly complete electrical circuits are not necessary. For example, only the motor requires a power supply, and every component has an automatic ground. Similarly, pull-up and pull-down resistors are not normally necessary, although they can be used for three-state logic, buses, etc. To do realistic modeling of electrical characteristics (e.g., with Ohm's Law) would be a matter of encoding this behavior into the Shelby programs of the affected components. In other words, conceptualization does not preclude deeper modeling and analysis. For a holistic view, however, such treatment is often too detailed, distracting, and unnecessary.

2.3.2 Evaluation

Most course assignments, especially in Computer Science, tend to focus on synthesis, or building a solution. The equally important counterpart, analysis, which determines the effectiveness of the solution, usually receives far less attention [1]. The evaluation stage of simulation in ShelbySim allows students to exercise a design over a wide range of possible inputs, events, or constraints, and especially to test combinations of inputs that they might not consider. It facilitates reproducible, controlled experiments that generate the raw data students need to demonstrate performance quantitatively. The typical process is as follows:

1. Students debug their initial design until it works *correctly*, although it need not work *well* at this point.

1The enclosing graphical user interface of the ShelbySim application is ordinary and uninteresting. For clarity, this figure is extracted from it, and connection labels and other descriptive elements are omitted.

2. For the inputs, they define possible discrete values or ranges with step resolutions. For example, switch *E* has the options {`on`, `off`}, whereas slider *L* ranges from 0 to 100%, and a 5% step from minimum to maximum might be appropriate. Values can also have probabilities associated with them. For example, breaker *B* might have a 0.1% chance of going from `closed` to `open` as the result of an abnormal event like a defect (not because its power limit was exceeded, which is a normal operational event).

3. Students run the simulation. For input sets without probability, ShelbySim automatically varies the inputs across all combinations and logs the corresponding results to structured text files. Students can carefully select subsets and export them to Microsoft Excel™, MATLAB™, gnuplot, or other applications. Figure 2 is a sample plot of the absolute difference between D_E and D_A as a function of an absolute percentage change in either *T* or *L* after one second. It demonstrates latency in achieving a steady state in this simplistic controller model.

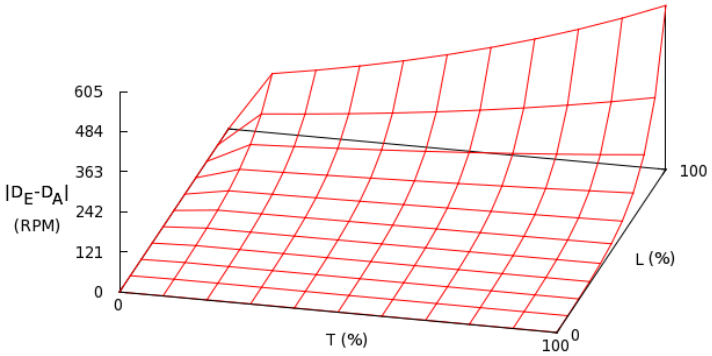

Figure 2: Sample output for analysis

For input sets with probability, a stochastic simulation occurs. Students select a number of runs, often between 100 and 10,000, depending on the complexity of the design and the resolution of its analysis. ShelbySim will perform the same process as above, but with independent run sets. Analysis must then take into account the distribution of results, which is much more complicated, but also much more representative of the real world. For example, the reliability of breaker *B* might be determined to be unacceptable if a large percent of the test runs failed due to it.

Stochastic simulation is especially useful for identifying emergent properties and subtle interdependencies between components that might not be considered for explicit testing [4]. For example, breaker *B*, and an additional breaker B_2 elsewhere, might individually have 0.1% failure rates, as expected. However, analysis might show that whenever *B* fails, B_2 fails 20% of the time.

Reproducibility stems from the way random numbers are managed in ShelbySim: for any given seed, the simulation will produce identical results. This capability allows students to re-examine individual runs that appear interesting or anomalous. For example, why, among 499 correct runs, did only number 73 produce the wrong answer? This frustrating scenario of diagnosing spurious results and transient failures is all too common in real life. ShelbySim helps students develop the skills and self-discipline to perform reasoning and analysis on the facts and assumptions to infer a cause, then to propose a remedy, and finally to test whether it actually resolves the problem. Too often, students haphazardly throw hardware or software at a problem, and it only coincidentally disappears, which is not an effective problem-solving strategy.

4. Either type of simulation in Step 3 establishes the baseline performance for the students' initial designs. Now they can experiment in a controlled manner to refine it. They iteratively modify one—and only one—input parameter, and then rerun the same simulation to produce a new set of results. When they compare this performance to the baseline performance, they see a direct cause-and-effect relationship with respect to that single modification. If performance improves, it sets them on a trajectory to improve it further. If it degrades, they use the rollback feature to undo the modification and try something else. This iterative process not only improves the design, it also improves the students' skills, wisdom, intuition, etc. at hypothesizing what might be promising modifications. Just as important, too, is that it dissuades them from undisciplined, labor-intensive, random trial-and-error experimentation and hacking.

Thanks to its flexible experimentation framework, ShelbySim is convenient for running hypothetical test cases, what-if scenarios, trade-off and cost-benefit analyses, and so on. It is also useful for determining performance limitations, reliability, security, etc., which contributes to optimizing and documenting designs. Furthermore, when students are given designs with intentional faults, it serves as a tool for diagnosing problems. In essence, ShelbySim helps students develop

sound systems-engineering practices and thought patterns by experiencing real operational contexts, as Figure 3 illustrates [5,6]:

a. Data: no associativity or context
b. Information: associativity within one context
c. Knowledge: associativity within multiple contexts
d. Wisdom: generalization of principles based on knowledge from different sources over time

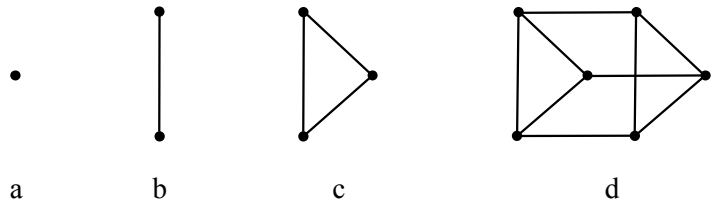

Figure 3: Development of engineering thought patterns

ShelbySim does not actually perform any data analysis or presentation. It only generates the structured output data for such activities. Ultimately, it is the students' responsibility to process the data and write up a report. Communication skills are woefully underaddressed in most engineering programs [7]. Most of the limited time available for each assignment is spent on synthesis, at the expense of analysis and reporting. By migrating much of the administrative overhead required to run experiments from the students to the software, ShelbySim makes room for analysis and communication-related aspects.

3 Recruitment Applications

According to anecdotal comments, Computer Science lacks a tangible "coolness" factor, perhaps because the age group of potential students has grown up with computers and considers them commonplace and uninteresting [8]. Fortunately, the colorful, action-packed, interactive displays in simulation mode serve as an attractive demonstration for career and majors fairs and related events. ShelbySim can play an important recruitment role, especially in light of the federal initiative to double STEM graduates by 2015. Many pre-college students, even those with technical interests, may not truly realize the widespread role computer-based systems play in nearly every modern device. Even if they do not ultimately consider a major in these areas, a minor might still be a persuasive option.

Younger groups are also a good target for demonstrations. An earlier, non-interactive version has been used at a number of events. It succeeded in drawing students to the College of Engineering table, where representatives fielded questions and related the exciting roles of engineering at the appropriate level. As a simple tool with substantial breadth and depth of appealing content, ShelbySim has great potential in the K-12 recruitment field.

4 Assessment

As a work in progress, ShelbySim is not yet available in its entirety for quantitative assessment. Substantial parts of it, however, have been successfully fielded as proofs of concept in senior/graduate-level courses in compilers and computer architecture. Student feedback was positive and encouraging.

Assessment as this point is based on the potential of ShelbySim to be a useful pedagogical tool. The American Association of Higher Education lists seven principles for good practice in undergraduate education [9]. ShelbySim aligns well with them.

4.1 Encourages Student-Faculty Contact

Components (and entire designs) are self-contained packages. If students have a question or problem, they can select a component, annotate it, and automatically send it via email to the instructor, who can import it without hassle, run it, make comments to it directly, and send it back for seamless reintegration. This capability streamlines the communication process and makes it more likely that students will seek assistance. When multiple students encounter a common problem —often the intended crux of an assignment—the instructor can email hints to the entire class or address it in lecture on his or her own ShelbySim.

4.2 Encourages Cooperation Among Students

For larger projects in a collaborative environment, teams for different components of a shared design may work relatively independently, once they have agreed on the interdependency and intercommunication specifications. The intent is to

avoid the undisciplined—yet seductive—process of collectively hacking everything together. While it is difficult to enforce this process, ShelbySim components can track their history throughout a project and determine who worked on what, when, and for how long, etc. This capability provides some after-the-fact documentation on the degree of cooperation and discipline.

The year-long Senior Design capstone course in the College of Engineering at Idaho State University requires multidisciplinary teams. Although ShelbySim has not been officially fielded in this context yet, faculty believe it has great potential.

4.3 Encourages Active Learning

ShelbySim is fun! Most students study engineering and computers because they have a passion for seeing and making things work. This software allows them to observe in a hands-on, virtual environment the details of how systems operate, and to play around with them freely. Open-ended assignments allow students to be creative by asking them to innovate, improve a design, and justify it, much the way the process works in the commercial world [10,11]. This perspective is learner-centered instead of teacher-centered [12].

4.4 Gives Prompt Feedback

Interactive simulation produces immediate results. Even without a test plan, students can initially see whether fiddling with the inputs causes failure. After achieving an operating solution, they can then determine whether the outputs are consistent with expectation and actually solve the task. And finally, they can evaluate how well the solution solves the task. All three steps are part of an iterative engineering process of refining a solution.

Batch simulation simplifies and expedites grading. Running the same tests, with the same random seeds, on all submissions ensures consistency. It also streamlines the process of submitting and returning work because ShelbySim packages everything for convenient electronic transfer: students submit their solution as a single email attachment or web-page upload, then the grader evaluates it, makes comments directly in it, and returns it electronically. There is no confusion about which files to submit, how to assemble them for grading, and so on. The less time the grader spends on administrative overhead, the more time he or she has to provide prompt and meaningful feedback.

4.5 Emphasizes Time on Task

ShelbySim can hide and expose details as appropriate. It separates stages in the engineering design process to focus students on relevant issues and to minimize distractions. It is also friendly, straightforward software that allows students to spend more time on the project than on the tool itself. In contrast, real-world, production-level software, when used in the classroom environment, tends to overwhelm students with a landslide of options that they do not need or understand.

4.6 Communicates High Expectations

The philosophy of ShelbySim is to make sound engineering design inherently part of the education process. The goal of its use is to avoid the typical assignment cycle where students hack requirements into something reasonably working at the last minute, submit it into a void, receive a grade and perhaps feedback, and never look at any of this again. This cycle is not conducive to developing and implementing a conscientious plan, then evaluating the results to understand what worked, what did not, and what could be improved, etc.

4.7 Respects Diverse Talents and Ways of Learning

ShelbySim is a visual tool, which lends itself well to visual learners. Kinesthetic learners should also relate well to it because it represents a three-dimensional world of circuits and physical devices in an animated, hands-on, virtual environment. Auditory learners might benefit from its use as a lecture support tool, especially for realtime, interactive demonstrations of concepts. Students taking engineering courses via distance learning or the web can benefit because they can run examples at their own pace away from campus, but the tool still allows them to communicate with the instructor and the rest of the class through the packaging feature.

5 Conclusion

ShelbySim's organization into components for software, hardware, and simulation maps well to the needs of a variety of courses in the curricula for computer science, computer engineering, electrical engineering, mechanical engineering, mechatronics, and measurements and controls. The components integrate seamlessly for a holistic view of entire systems, but they can also function relatively standalone for more localized emphasis. A hallmark of ShelbySim is its unified framework for implementing a design, simulating it in a variety of operational contexts, and generating the structured data

for external analysis, evaluation, and reporting. Its goal is to foster critical-thinking skills for both the synthesis and analysis stages in engineering. In particular, the former aligns with traditional assignments to build something, whereas the latter addresses underemphasized analytical, diagnostic, and communication skills. ShelbySim mitigates much of the mundane overhead in assignments to provide students with more hands-on time with the content.

As a work in progress, ShelbySim is still undergoing significant development. Once it becomes stable for public release, and its documentation is complete, it will be freely available and continually supported on the author's web site at `www.isu.edu/~tappdan`. The source code will also be available for inspection and modification under the terms of the GNU General Public License.

6 References

[1] Computing Curricula 2001 project (CC2001), final report, Association for Computing Machinery, December 2001.

[2] American Competitiveness Initiative, U.S. Department of Education, http://www.ed.gov/about/inits/ed/competitiveness/index.html, checked 13 November 2008.

[3] Devasia, S. and Meek, S. PC's and Micro-Controllers in Mechatronics Education. In proceedings of Frontiers in Education, FIE96, 6-9 Nov. 1996, pp. 966-969.

[4] Casadei, M.; Gardelli, L.; and Viroli, M. Simulating Emergent Properties of Coordination in Maude: the Collective Sort Case. Electronic Notes in Theoretical Computer Science, Vol. 175, Issue 2, pp. 59-80, June 2007.

[5] Pressman, R. Software Engineering: A Practitioner's Approach, 6th ed., 2004. McGraw-Hill: New York.

[6] Irish, R. Engineering Thinking: Using Benjamin Bloom and William Perry to Design Assignments, Language and Learning Across the Disciplines, Vol. 3, no. 2, pp. 83-102, 1999.

[7] Tong, L. Identifying essential learning skills in students' engineering education. In proceedings of HERDSA 2003, Canterbury, New Zealand.

[8] Jonsson, P. "Can competitions raise 'cool' factor of math, science?" Christian Science Monitor, 17 May 2008.

[9] Chickering, A. and Gamson, Z. "Seven principles of good practice in Undergraduate Education." AAHE Bulletin, 39, pp. 3-7, 1987.

[10] Wiesner, T. and Lan, W. Comparison of Student Learning in Physical and Simulated Unit Operations Experiments, Journal of Engineering Education, July 2004.

[11] Dilli, Z.; Goldsman, N.; Schmidt, J.; Harper, L.; and Marcus, S. A New Pedagogy in Electrical and Computer Engineering: An experiential and conceptual approach. In proceedings of Frontiers in Education, FIE02, 6-9 Nov. 2002.

[12] Yearny, M. Teacher-Centered to Learner-Centered Educational Model. In proceedings of Frontiers in Education, FIE98, 4-7 Nov. 1998.

Biographical Information

DAN TAPPAN

Dan Tappan is an Assistant Professor in the Department of Computer Science at Idaho State University. He received his Ph.D. in Computer Science from New Mexico State University in 2004 and his MSE in Computer Systems Engineering from the University of Arkansas at Fayetteville in 1996. His research interests are artificial intelligence, especially natural-language processing, as well as modeling and simulation, and pedagogy of the STEM (science, technology, engineering, and mathematics) disciplines.